The Journal of Systems & Software 144 (2018) 106–120

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Controversy Corner

Investigating faults missed by test suites achieving high code coverage

Amanda Schwartz*,[a], Daniel Puckett[a], Ying Meng[b], Gregory Gay[b]

[a] University of South Carolina Upstate, Spartanburg, SC, United States
[b] University of South Carolina, Columbia, SC, United States

ARTICLE INFO

Keywords:
Code coverage
Automated testing
Software testing
Test suite effectiveness

ABSTRACT

Code coverage criteria are commonly used to determine the adequacy of a test suite. However, studies investigating code coverage and fault-finding capabilities have mixed results. Some studies have shown that creating test suites to satisfy coverage criteria has a positive effect on finding faults, while other studies do not. In order to improve the fault-finding capabilities of test suites, it is essential to understand what is causing these mixed results. In this study, we investigated one possible source of variation in the results observed: fault type. Specifically, we studied 45 different types of faults and evaluated how effectively human-created test suites with high coverage percentages were able to detect each type of fault. Our results showed, with statistical significance, there were specific types of faults found less frequently than others. However, improvements in the formulation and selection of test oracles could overcome these weaknesses. Based on our results and the types of faults that were missed, we suggest focusing on the strength of test oracles along with code coverage to improve the effectiveness of test suites.

1. Introduction

In order to ensure software quality, it is essential that the software is tested thoroughly. However, what exactly constitutes *thorough* testi ng can be subjective. As developers lack knowledge of the faults that may reside in their systems, guidance and a means of judging test suite *adequacy* is required. Currently, one of the most popular ways to evaluate the adequacy of a test suite is through the use of code coverage criteria.

Code coverage criteria evaluate test suites by examining how well they cover structural elements such as functions, statements, branches, and/or conditions of a software system (Pezze and Young, 2006). Each criterion establishes a set of test obligations over the class-under-t est (CUT) that must be fulfilled in order to satisfy the criterion. Coverage criteria are commonly used in both academic research and industry, as they are easy to understand, establish clear guidelines and stopping conditions for testing, and are well-supported across a variety of programming languages (Groce et al., 2014). Consequently, the confidence in code coverage being a proper method of evaluating test suites has become high in the software testing community. In fact, the confidence is so high that in some domains, such as avionics, evaluating test suites through the use of code coverage is legally required RTCA/DO-178C, Heimdahl et al. (2008). Many research studies also validate proposed techniques by their ability to achieve some level of code coverage (e.g. Artzi et al., 2011; Yang et al., 2007).

Contrary to the widespread use and acceptance of code coverage being an adequate measure of test suite effectiveness, studies investigating the relationship between code coverage and fault-finding capabilities do not consistently support this. Some studies have shown that generating test suites to satisfy code coverage criteria has a positive effect on finding faults (e.g. Gopinath et al., 2014; Namin and Andrews, 2009; Gay, 2017), while other studies do not (e.g. Gay et al., 2015b; Inozemtseva and Holmes, 2014). To better evaluate whether code coverage is a proper method of evaluating test suites, it is important to understand why there are such differences in these findings.

In order to accept high code coverage as an indicator for a test suite's ability to find faults, there should be consistent evidence that test suites with high code coverage are capable of finding more faults than test suites with lower code coverage. However, since the research does not always support this, it is important to investigate whether there are particular factors that affect the ability of a test suite achieving high code coverage to find faults. Unfortunately, very little work has been done in this area. Few studies were found (Gay et al., 2016; Heimdahl et al., 2008; Zhang and Mesbah, 2015) that identify factors that influence the relationship between code coverage and fault detection. These studies provide some insight—particularly around the influence of program structure—but cannot completely explain the different findings in the studies investigating code coverage and fault detection. More research needs to be conducted to investigate this important issue.

In our previous work (Schwartz and Hetzel, 2016), we began to

* Corresponding author.
E-mail addresses: aschwar2@uscupstate.edu (A. Schwartz), dpuckett@email.uscupstate.edu (D. Puckett), ymeng@email.sc.edu (Y. Meng), greg@greggay.com (G. Gay).

https://doi.org/10.1016/j.jss.2018.06.024
Received 20 April 2017; Received in revised form 4 June 2018; Accepted 7 June 2018

investigate the impact particular *fault types* had on the relationship between code coverage and fault finding effectiveness, as modeled through the use of mutation testing—the seeding of synthetic faults into the CUT. Specifically, we were interested in whether there were particular fault types that went undetected more frequently than other fault types when programs are evaluated by test suites that achieve high code coverage. Our research showed that the rate of fault detection varied significantly according to fault type. We also noticed that there were certain types of faults consistently found less frequently than others. These were interesting findings that could inspire future research on why these particular fault types were found less frequently. However, this study was limited in two ways. First, only class-level mutation operators were considered. Second, there were many mutation operators that did not produce enough mutants to have enough data to perform a statistical analysis or form any solid conclusions. We address these limitations in this paper. Specifically, this paper makes the following contributions:

- The paper is extended to consider 19 Traditional Mutant Operators in addition to the original 26 Class Level Mutation Operators considered in our previous work (Schwartz and Hetzel, 2016).
- An additional 25,100 Class-Level mutants were created to supplement the 15,834 Class-Level mutants created in our previous work, for a total of 40,934 Class-Level mutants.
- A total of 122,985 Traditional mutants were created and analyzed.
- Statistical tests were performed and presented to determine whether there is a statistical significance to the faults that go undetected more frequently than other faults.
- A discussion of the fault types identified as outliers by the statistical tests is included.
- A suggestion, based on the nature of the faults identified as outliers, on how to improve test suites and test suite evaluation techniques is provided.

Our results identified that two types of mutants were found disproportionately often—AORB (Arithmetic Operator Replacement) and ROR (Relational Operator Replacement). However, four types of mutants were found less often than expected: AOIS (Arithmetic Operator Insertion), PCI (Type Case Operator Insertion), EAM (Accessor Method Change), and AODU (Arithmetic Operator Deletion). Test oracles that more thoroughly inspect internal state would aid in revealing such faults. Ultimately, code coverage alone does not ensure that faults are triggered and detected, and the selection of input and oracle have a dual influence on the effectiveness of a test suite. More attention should be given to the thoroughness of the selected oracle, and to the variables that are monitored and checked by the oracle.

The rest of the paper is organized as follows. Section 2 presents background information and related work. Section 3 explains our experimental procedures. Section 4 presents the results of our experiment. A discussion of our results is presented in Section 5. And finally, conclusions are discussed in Section 6.

2. Background and related work

Software testing is extremely important to the development process as the means of ensuring that software has the correct functionality and is not defective. However, software testing can be very time consuming and costly. Therefore, a significant amount of time and effort has been spent to identify ways to reduce the cost of software testing. Much of this attention has been spent researching automated software testing procedures. As a result, many different testing platforms and methods have been proposed, developed, and evaluated (e.g. Anand et al., 2013; Jensen et al., 2013; Nguyen et al., 2014) and a great deal of progress has been made on reducing the time necessary for software testing. By reducing time, the hope is it will also reduce cost. However, the cost will only be reduced if testing continues to be effective at finding faults. Missed faults are extremely costly, and any reduction in cost resulting from the decreased time would be lost with the increase of costs associated with missed faults. Therefore, automated testing methods need to be evaluated to be sure they are effective at finding faults.

The most commonly used metric to evaluate automated test suites is to use some form of code coverage criteria. Coverage criteria reports a percentage according to how much source code is executed by the test suites. The exact calculation depends on the coverage method used. For example, line coverage is a very simple coverage metric that simply reports the percentage of lines of code that are covered by the test suite. Another popular coverage metric, branch coverage, adds an additional requirement that at each conditional both the true path and the false path will be executed.

Coverage criteria has become widely accepted in the software testing community as an adequate measure of test suite effectiveness (Groce et al., 2014). It is used to validate new automated testing methods (e.g. Fraser and Arcuri, 2011; Ghosh et al., 2013; Marinescu and Cadar, 2013), used to compare testing methods based on level of coverage (e.g. Amalfitano et al., 2012; Gross et al., 2012; Inkumsah and Xie, 2008), and is used in many domains to determine whether a test suite as adequate. In fact, in some safety cricital domains, such as avionics, code coverage is legally required to determine the adequacy of a test suite RTCA/DO-178C.

Since coverage criteria is frequently the standard for many in terms of evaluating test suites, it is important to make sure code coverage *actually* is a good indicator of test suite effectiveness. Some recent research has been conducted to evaluate whether increasing code coverage also increases a test suite's ability to find faults. The results of this research has been mixed. Some studies show achieving high code coverage is a good indicator for fault-finding capabilities, while other studies do not.

Studies which provide support for coverage criteria being an adequate measure of test suite effectiveness show a correlation between code coverage and fault-finding capabilities. For example, early work by Frankl and Weiss (1993) shows a correlation between test suite effectiveness and all-use and decision coverage criteria. Cai and Lyu (2005) found a moderate correlation between fault-finding capabilities and four different code coverage criteria. Del Frate et al. (1995) report a study which finds a higher correlation between test suite effectiveness and block coverage than there is between test suite effectiveness and test suite size. Gligoric et al. (2013) studied a total of 26 programs and found that test suite effectiveness was correlated with coverage, and reported branch coverage as performing the best. A recent study by Kochhar et al. (2015) evaluated two industrial programs and found a correlation between code coverage and faults detected. In our past work examining the factors that indicated a high likelihood of fault detection, we found that high levels of code coverage had a stronger correlation to the likelihood of fault detection than the majority of the other measured factors (Gay, 2017). However, we also found that coverage alone was not enough to ensure fault detection.

Even though there are a number of studies that show a correlation between code coverage and fault-finding effectiveness, there are also many studies which do not. In previous work, we reported that satisfying code coverage alone was a poor indication of test suite effectiveness when suites are generated specifically to achieve coverage (Gay et al., 2015b; Staats et al., 2012). We studied the fault-finding effectiveness of automatically generated test suites that satisfied five code coverage criteria (branch, decision, condition, MC/DC, and Observable MC/DC) and compared them to randomly generated test suites of the same size for five different production avionics systems. We found that, for most criteria, test suites automatically generated to achieve coverage performed *significantly worse* than random test suites of equal size which did not work to achieve high coverage. We did find that coverage had some utility as a *stopping criterion*, however. Randomly-generated test suites that used coverage as a stopping criterion outperformed equally-sized suites generated purely randomly. The